计算机组成

机器语言(2)

高小鹏

北京航空航天大学计算机学院
系统结构研究所

Review of Last Lecture

- RISC Design Principles
  - Smaller is faster: 32 registers, fewer instructions
  - Keep it simple: rigid syntax, fixed word length
- MIPS Registers: `$s0-$s7`, `$t0-$t9`, `$0`
  - Only operands used by instructions
  - No variable types, just raw bits
- Memory is byte-addressed
  - Watch endianness when dealing with bytes

Review of Last Lecture

- MIPS Instructions
  - Arithmetic: `add, sub, addi, mult, div` `addu, subu, addiu`
  - Data Transfer: `lw, sw, lb, sb, lbu`
  - Branching: `beq, bne, j`
  - Bitwise: `and, andi, or, ori, nor, xor, xori`
  - Shifting: `sll, sllv, srl, srlv, sra, srav`

Levels of Representation/Interpretation

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., MIPS)

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

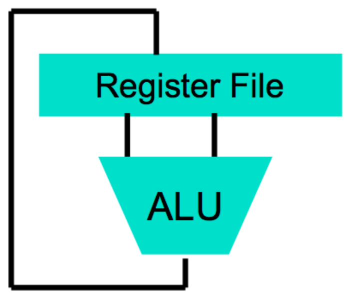
Assembler

Machine Language Program (MIPS)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

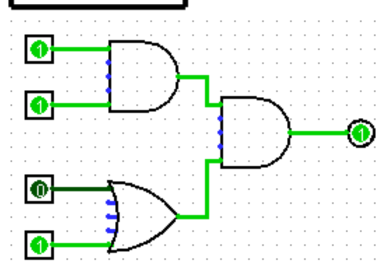
Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



提纲

- 内容主要取材：CS61C的6讲
  - http://inst.eecs.berkeley.edu/~cs61c/su12
- 不等式
- 伪指令
- 实现函数
- 函数调用约定

提纲

- 内容主要取材：CS61C的6讲
  - http://inst.eecs.berkeley.edu/~cs61c/su12
- 不等式
- 伪指令
- 实现函数
- 函数调用约定

Inequalities in MIPS

加上=，总共多少种判断？

- Inequality tests: <, <=, >, and >=
  - RISC: implement all with 1 additional instruction
- Set on Less Than (`slt`)
  - `slt dst,src1,src2`
  - Stores 1 in `dst` if value in `src1` < value in `src2` and stores 0 in `dst` otherwise
- Combine with `bne`, `beq`, and `$0`

设计权衡：
3指令方案： ISA小；2条指令/判断；单条性能可能好点点
5指令方案： ISA大；1条指令/判断；单条性能可能差点点

总性能？

Inequalities in MIPS

- C Code:

```
if (a < b) {
   ... /* then */
}
```

(let a→$s0, b→$s1)

- MIPS Code:

```
slt $t0,$s0,$s1
# $t0=1 if a<b
# $t0=0 if a>=b
bne $t0, $0,then
# go to then
#    if $t0≠0
```

Inequalities in MIPS

- C Code:

```
if (a >= b) {
  ... /* then */
}
```

(let `a`→`$s0`, `b`→`$s1`)

- MIPS Code:

```
slt $t0,$s0,$s1
# $t0=1 if a<b
# $t0=0 if a>=b
beq $t0, $0,then
# go to then
#   if $t0=0
```

- Try to work out the other two on your own:
  - Swap `src1` and `src2`
  - Switch `beq` and `bne`

Immediates in Inequalities

- Three variants of `slt`:
  - `sltu  dst,src1,src2`: unsigned comparison
  - `slti  dst,src,imm`: compare against constant
  - `sltiu dst,src,imm`: unsigned comparison against constant
- Example:

```
addi  $s0,$0,-1  # $s0=0xFFFFFFFF
slti  $t0,$s0,1  # $t0=1
sltiu $t1,$s0,1  # $t1=0
```

Aside: MIPS Signed vs. Unsigned

- MIPS terms "signed" and "unsigned" appear in 3 different contexts:
  - Signed vs. unsigned bit extension
    - `lb`
    - `lbu`
  - Detect vs. don't detect overflow
    - `add, addi, sub, mult, div`
    - `addu, addiu, subu, multu, divu`
  - Signed vs. unsigned comparison
    - **`slt, slti`**
    - **`sltu, sltiu`**

Question: What C code properly fills in the following blank?

```
do {i--;} while(__________);
```

```
Loop:                    # i→$s0, j→$s1
addi $s0,$s0,-1          # i = i - 1
slti $t0,$s1,2           # $t0 = (j < 2)
beq  $t0,$0 ,Loop        # goto Loop if $t0==0
slt  $t0,$s1,$s0         # $t0 = (j < i)
bne  $t0,$0 ,Loop        # goto Loop if $t0!=0
```

- ☐ $\boxed{j \geq 2 \mid\mid j < i}$
- ☐ j ≥ 2 && j < i
- ☐ j < 2 || j ≥ i
- ☐ j < 2 && j ≥ i

提纲

- 内容主要取材：CS61C的6讲
  - http://inst.eecs.berkeley.edu/~cs61c/su12
- 不等式
- 伪指令
- 实现函数
- 函数调用约定

Assembler Pseudo-Instructions

- Certain C statements are implemented unintuitively in MIPS
  - e.g. assignment (`a=b`) via addition with `0`
- MIPS has a set of "pseudo-instructions" to make programming easier
  - More intuitive to read, but get translated into actual instructions later
- Example:

```
move dst,src
```
translated into
```
addi dst,src,0
```

Assembler Pseudo-Instructions

- List of pseudo-instructions: http://en.wikipedia.org/wiki/MIPS_architecture#Pseudo_instructions
  - List also includes instruction translation
- Load Address (`la`)
  - `la dst,label`
  - Loads address of specified label into `dst`
- Load Immediate (`li`)
  - `li dst,imm`
  - Loads 32-bit immediate into `dst`
- MARS has additional pseudo-instructions
  - See Help (F1) for full list

Assembler Register

- Problem:
  - When breaking up a pseudo-instruction, the assembler may need to use an extra register
  - If it uses a regular register, it'll overwrite whatever the program has put into it
- Solution:
  - Reserve a register ($1 or $at for "assembler temporary") that assembler will use to break up pseudo-instructions
  - Since the assembler may use this at any time, it's not safe to code with it

编号	名称	用途
0	$zero	常量0
1	$at	汇编器保留
2-3		
4-7		
8-15	$t0-$t7	临时变量
16-23	$s0-$s7	程序变量
24-25	$t8-$t9	临时变量
28		
29		
30		
31		

MAL vs. TAL

- True Assembly Language (TAL)
  – The instructions a computer understands and executes
- MIPS Assembly Language (MAL)
  – Instructions the assembly programmer can use (includes pseudo-instructions)
  – Each MAL instruction becomes 1 or more TAL instruction
- TAL ⊂ MAL

提纲

- 内容主要取材：CS61C的6讲
  - http://inst.eecs.berkeley.edu/~cs61c/su12
- 不等式
- 伪指令
- 实现函数
- 函数调用约定

Six Steps of Calling a Function

1. Put *arguments* in a place where the function can access them
2. Transfer control to the function
3. The function will acquire any (local) storage resources it needs
4. The function performs its desired task
5. The function puts *return value* in an accessible place and cleans up (restores any used registers)
6. Control is returned to you

MIPS Registers for Function Calls

- Registers way faster than memory, so use them whenever possible
- `$a0–$a3`: four *argument* registers to pass parameters
- `$v0–$v1`: two *value* registers to return values
- `$ra`: *return address* register that saves where a function is called from

编号	名称	用途
0	$zero	常量0
1	$at	汇编器保留
2-3	$v0-$v1	返回值
4-7	$a0-$a3	参数
8-15	$t0-$t7	临时变量
16-23	$s0-$s7	程序变量
24-25	$t8-$t9	临时变量
28		
29		
30		
31	$ra	返回地址

MIPS Instructions for Function Calls

- Jump and Link (`jal`)
  - `jal label`
  - Saves the location of *following* instruction in register `$ra` and then jumps to `label` (function address)
  - Used to invoke a function
- Jump Register (`jr`)
  - `jr src`
  - Unconditional jump to the address specified in `src` (almost always used with `$ra`)
  - Used to return from a function

Instruction Addresses

- `jal` puts the *address* of an instruction in `$ra`
- Instructions are stored as data in memory!
  - **Recall:** Code section
  - More on this next lecture
- In MIPS, all instructions are 4 bytes long so each instruction differs in address by 4
  - **Recall:** Memory is byte-addressed
- Labels get converted to instruction addresses

Program Counter

- The program counter (PC) is a special register that holds the address of the current instruction being executed
  - This register is inaccessible to the programmer, but accessible to `jal`
- `jal` stores `PC+4` into `$ra`
  - What would happen if we stored `PC` instead?
- All branches and jumps (`beq`, `bne`, `j`, `jal`, `jr`) work by storing an address into `PC`

Function Call Example

```
... sum(a,b); ...          /* a→$s0,b→$s1 */

int sum(int x, int y) {
  return x+y;
}
```

C

MIPS

address (decimal)

```
1000  addi $a0,$s0,0        # x = a
1004  addi $a1,$s1,0        # y = b
1008  addi $ra,$zero,1016   # $ra=1016
1012  j    sum              # jump to sum
1016
...
2000  sum: add $v0,$a0,$a1
2004  jr   $ra              # return
```

Would we know this before compiling?

Otherwise we don't know where we came from

Function Call Example

```
... sum(a,b); ...            /* a→$s0,b→$s1 */

int sum(int x, int y) {
  return x+y;
}
```

C

MIPS

address (decimal)

```
1000   addi $a0,$s0,0         # x = a
1004   addi $a1,$s1,0         # y = b
1008   jal  sum               # $ra=1012, goto sum
1012
...

2000   sum: add $v0,$a0,$a1
2004   jr   $ra               # return
```

Six Steps of Calling a Function

1. Put *arguments* in a place where the function can access them `$a0-$a3`
2. Transfer control to the function `jal`
3. The function will acquire any (local) storage resources it needs
4. The function performs its desired task `$v0-$v1`
5. The function puts *return value* in an accessible place and cleans up (restores any used registers)
6. Control is returned to you `jr`

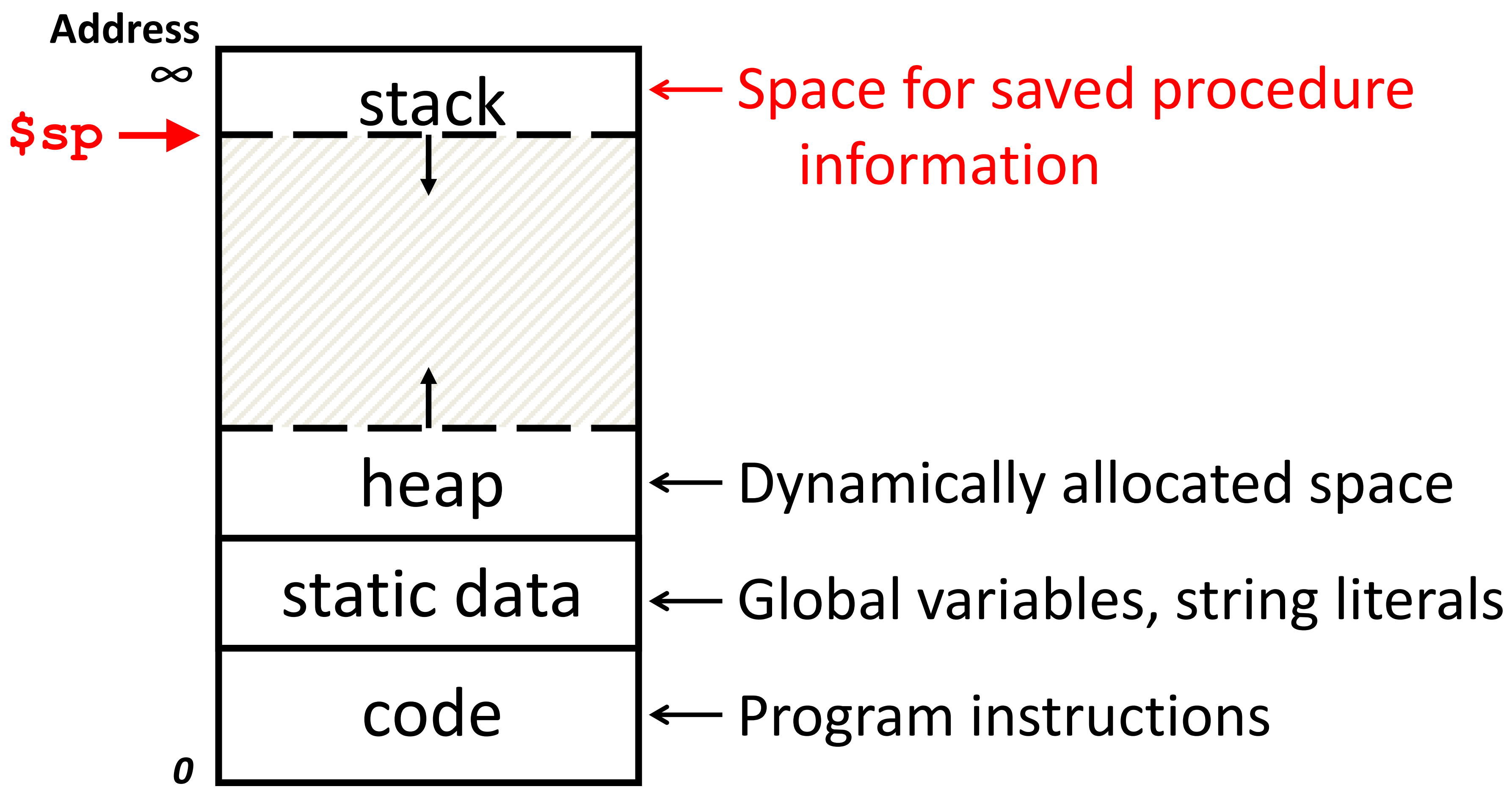
Saving and Restoring Registers

- Why might we need to save registers?
  - Limited number of registers for everyone to use
  - What happens if a function calls another function?

    (`$ra` would get overwritten!)
- Where should we save registers? The Stack
- `$sp` (stack pointer) register contains pointer to current bottom (last used space) of stack

编号	名称	用途
0	$zero	常量0
1	$at	汇编器保留
2-3	$v0-$v1	返回值/结果
4-7	$a0-$a3	参数
8-15	$t0-$t7	临时变量
16-23	$s0-$s7	程序变量
24-25	$t8-$t9	临时变量
28		
29	$sp	栈指针
30		
31	$ra	返回地址

Recall: Memory Layout



Example: sumSquare

```
int sumSquare(int x, int y) {
   return mult(x,x)+ y; }
```

- What do we need to save?
  - Call to `mult` will overwrite `$ra`, so save it
  - Reusing `$a1` to pass 2nd argument to `mult`, but need current value (`y`) later, so save `$a1`
- To save something to the Stack, move `$sp` down the required amount and fill the created space

Example: sumSquare

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y; }
```

```
sumSquare:
         addi $sp,$sp,-8     # make space on stack
"push"   sw $ra, 4($sp)      # save ret addr
         sw $a1, 0($sp)      # save y
         add $a1,$a0,$zero   # set 2nd mult arg
         jal mult            # call mult
         lw $a1, 0($sp)      # restore y
"pop"    add $v0,$v0,$a1     # ret val = mult(x,x)+y
         lw $ra, 4($sp)      # get ret addr
         addi $sp,$sp,8      # restore stack
         jr $ra
mult:    ...
```

Basic Structure of a Function

Prologue

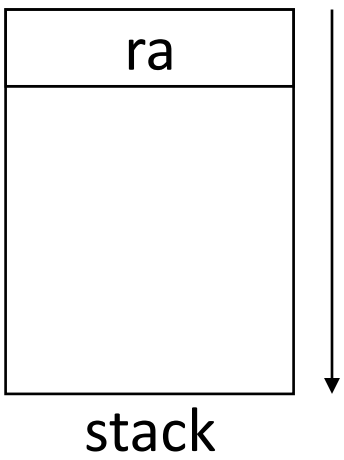
```
func_label:
addi $sp,$sp, -framesize
sw $ra, [framesize-4]($sp)
save other regs if need be
```

Body **(call other functions…)**

```
...
```

Epilogue

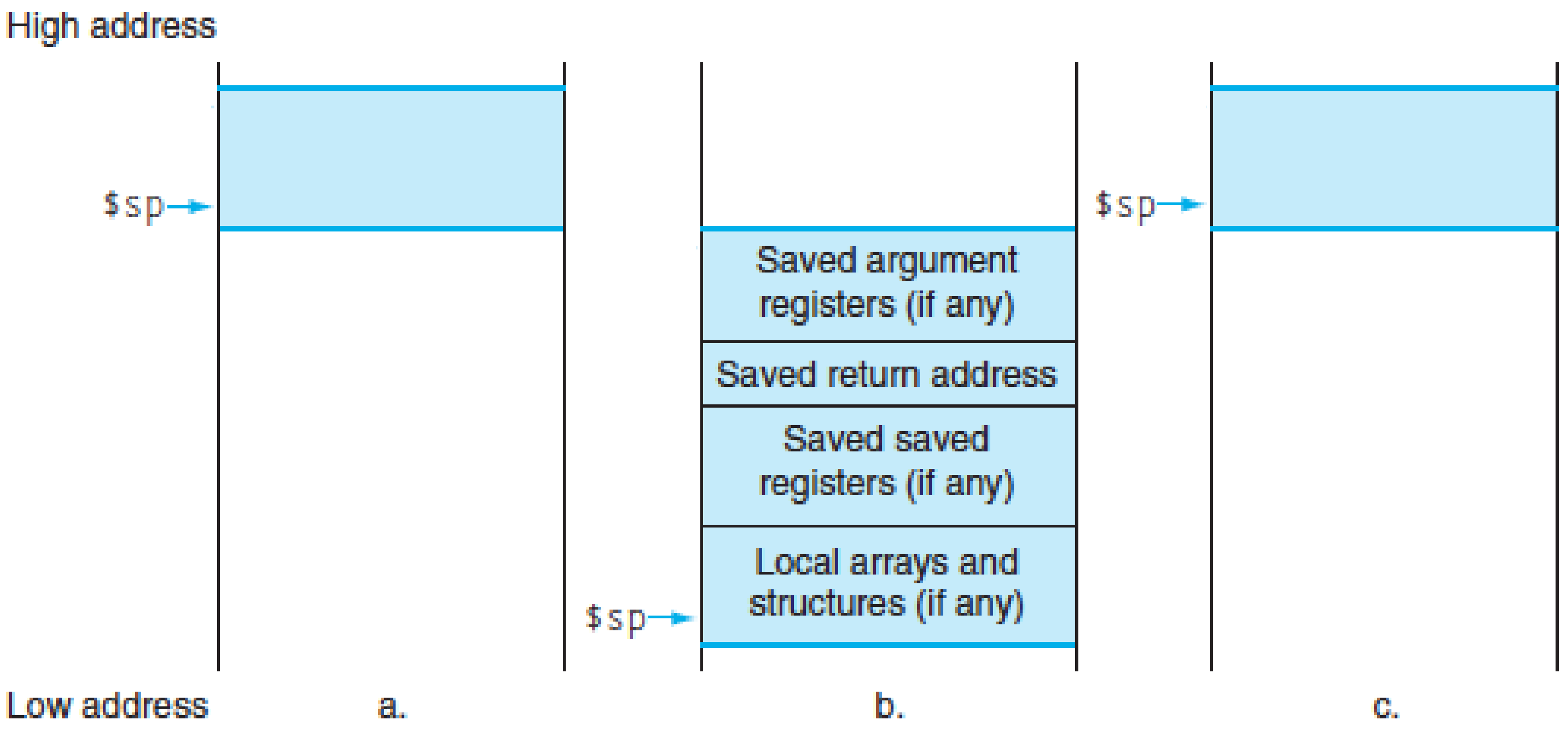
```
restore other regs if need be
lw $ra, [framesize-4]($sp)
addi $sp,$sp, framesize
jr $ra
```



Local Variables and Arrays

- Any local variables the compiler cannot assign to registers will be allocated as part of the stack frame (**Recall:** spilling to memory)
- Locally declared arrays and structs are also allocated as part of the stack frame
- Stack manipulation is same as before
  - Move `$sp` down an extra amount and use the space it created as storage

Stack Before, During, After Call



提纲

- 内容主要取材：CS61C的6讲
  - http://inst.eecs.berkeley.edu/~cs61c/su12
- 不等式
- 伪指令
- 实现函数
- 函数调用约定

Register Conventions

- **Calle<u>R</u>:** the calling function
- **Calle<u>E</u>:** the function being called

- Register Conventions: A set of generally accepted rules as to which registers will be unchanged after a procedure call (`jal`) and which may have changed

Saved Registers

- These registers are expected to be the same before and after a function call
  - If calleE uses them, must restore values before returning
  - This means save the old values, use the registers, then reload the old values back into the registers
- `$s0-$s7` (*saved* registers)
- `$sp` (stack pointer)
  - If not in same place, the caller won't be able to properly restore values from the stack
- `$ra` (return address)

Volatile Registers

- These registers can be freely changed by the calleE
  - If calleR needs them, must save values before making procedure call
- `$t0-$t9` (*temporary* registers)
- `$v0-$v1` (return values)
  - These will contain the new returned values
- `$a0-$a3` (arguments)
  - These will change if calleE invokes another function (nested function means calleE is also a calleR)

Register Conventions Summary

- One more time for luck:
  - CalleR must save any volatile registers it is using onto the stack before making a procedure call
  - CalleE must save any saved registers it intends to use before garbling up their values
- Notes:
  - CalleR and calleE only need to save the appropriate registers *they are using* (not all!)
  - Don't forget to restore the values later

Example: Using Saved Registers

```
myFunc: # Uses $s0 and $s1
   addiu     $sp,$sp,-12 # This is the Prologue
   sw        $ra,8($sp)  # Save saved registers
   sw        $s0,4($sp)
   sw        $s1,0($sp)
   ...                   # Do stuff with $s0 and $s1
   jal       func1       # $s0 and $s1 unchanged by
   ...                   #   function calls, so can keep
   jal       func2       #   using them normally
   ...                   # Do stuff with $s0 and $s1
   lw        $s1,0($sp)  # This is the Epilogue
   lw        $s0,4($sp)  # Restore saved registers
   lw        $ra,8($sp)
   addiu     $sp,$sp,12
   jr        $ra         # return
```

Example: Using Volatile Registers

```
myFunc: # Uses $t0
   addiu      $sp,$sp,-4   # This is the Prologue
   sw         $ra,0($sp)   # Save saved registers
   ...                     # Do stuff with $t0
   addiu      $sp,$sp,-4   # Save volatile registers
   sw         $t0,0($sp)   #   before calling a function
   jal        func1        # Function may change $t0
   lw         $t0,0($sp)   # Restore volatile registers
   addiu      $sp,$sp,4    #   before you use them again
   ...                     # Do stuff with $t0
   lw         $ra,0($sp)   # This is the Epilogue
   addiu      $sp,$sp,4    # Restore saved registers
   jr         $ra          # return
```

Choosing Your Registers

- Minimize register footprint
  - Optimize to reduce number of registers you need to save by choosing which registers to use in a function
  - Only save when you absolutely have to
- Function does NOT call another function
  - Use only `$t0-$t9` and there is nothing to save!
- Function calls other function(s)
  - Values you need throughout go in `$s0-$s7`, others go in `$t0-$t9`
  - At each function call, check number arguments and return values for whether you or not you need to save

Question: Which statement below is FALSE?

- MIPS uses `jal` to invoke a function and `jr` to return from a function
- `jal` saves `PC+1` in `$ra`
- The callee can use temporary registers (`$t`*i*) without saving and restoring them
- The caller can rely on save registers (`$s`*i*) without fear of callee changing them

Summary (1/2)

- Inequalities done using `slt` and allow us to implement the rest of control flow
- Pseudo-instructions make code more readable
  - Count as MAL, later translated into TAL
- MIPS function implementation:
  - Jump and link (`jal`) invokes, jump register (`jr $ra`) returns
  - Registers `$a0-$a3` for arguments, `$v0-$v1` for return values

Summary (2/2)

- Register conventions preserves values of registers between function calls
  - Different responsibilities for calleR and calleE
  - Registers classified as saved and volatile
- Use the Stack for spilling registers, saving return address, and local variables

作业

- 《计算机组成与设计》
  - WORD：2.8，2.10，2.11，2.12，2.15，2.19.1/3，2.20，2.21，2.23
- 竞赛：1～100倒序的指令数
  - 直接在汇编的data段中定义100个字节，依次存放1~100
  - 提交asm和报告
  - 建议在